VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SADA PROCEDURÁLNĚ GENEROVANÝCH EFEKTŮ

SET OF PROCEDURALLY GENERATED EFECTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MAREK BARVÍŘ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. TOMÁŠ STARKA**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/20941/2017/xbarvi00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Barvíř Marek**
Obor: Informační technologie
Téma: **Sada procedurálně generovaných efektů**
Set of Procedurally Generated Efects
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte teorii procedurtalniho generování.
2. Seznamte se s OpenGL, GPUEnginem, příp. dalšími knihovnami potřebnými pro práci.
3. Navrhněte a implementujte sadu procedurálně generovaných efektů, jako knihovnu/modul
4. Udělejte demonstrační aplikaci.
5. Vytvořte krátké video, prezentující vaši práci.

Literatura:

- Dohodou

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a část bodu 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Starka Tomáš, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Tato bakalářská práce se zabývá vytvořením knihovny pro procedurálně generované efekty. Popisuje techniky, využívané při vytváření navržených efektů, kterými jsou: laser, elektrický oblouk, energetická střela a štít. Výsledkem je demonstrační aplikace využívající navržené knihovny s předpřipravenou scénou.

Abstract

This bachelor thesis deals with creation of a library for procedural generated effects. The paper describes techniques used for creating effects such as: laser, electric arc, energetic shot and shield. The result is a demonstration application which uses designed libraries with pre-prepared scene.

Klíčová slova

OpenGL, GLSL, GPUEngine, procedurální generování, Perlinův Šum, Simplexní Šum, skybox, billboarding

Keywords

OpenGL, GLSL, GPUEngine, procedural generation, Perlin Noise, Simplex Noise, skybox, billboarding

Citace

BARVÍŘ, Marek. *Sada procedurálně generovaných efektů.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Starka

Sada procedurálně generovaných efektů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Starky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Marek Barvíř
16. května 2018

Poděkování

Rád bych poděkoval vedoucímu Ing. Tomášovi Starkovi za jeho rady a připomínky.

Obsah

Kapitola 1

Úvod

V rámci této bakalářské práce jsem navrhnul 4 proceduralní efekty. První dva efekty jsou různé typy střel ve tvaru laseru a energetické koule. Dalším je energetický štít, který je za normalních okolností slabě viditelný a po nárazu střely se na štítu objevuje pulzní vlna, která přejede přes celý štít a zviditelní jej. Tyto tři efekty, by mohly být použity při hře či grafickém intru s motivy vesmíru. Posledním efektem je elektrický oblouk, který může být základem pro kouzlo nebo část blesku.

Obecně procedurální generování je vytváření obsahu za pomocí algoritmů. Změnou jednotlivých vstupních parametrů získáváme unikátní obsah, minimálně pro každou novou kombinaci. Unikátnosti se dosahuje prostřednictvím generatorů pseudonáhodných čísel, šumů, L-systémů a mnohých další metod. Primární uplatnění procedurální grafiky je v počítačových hrách.

Dříve se procedurální grafika využívala primárně kvůli omezené paměti počítačů. Kvůli tomuto omezení byl obsah jako textury, nepřátelé a mapy algoritmicky generován za běhu programu. Díky tomu mohly hry velice navýšit znovuhratelnost pomocí generování místností, nepřátel a dalšího obsahu. Často se využívalo předdefinovaných seedů pro generaci stejného objektu, aby nemusel zabírat místo v paměti.

V současnosti počítače nejsou omezeny pamětí, ale i přesto jsou procedurální efekty stále neodělitelnou součástí počítačové grafiky. Často se využívají pro věci, kterých se nachazí ve hře mnoho a bylo by je nemožné vymodelovat v takovém počtu. Například miliony planet a nebo by modelování trvalo příliš dlouhou dobu.

Kapitola 2

Teorie

V této kapitole jsou popsány jednotlivé techniky, které jsou využity při vytváření procedurálně generované grafiky a níže zmíněných efektů.

2.1 Generování náhodných čísel

Generátory náhodných čísel mají v informatice velké uplatnění především v kryptografii, simulaci, jsou také neodělitelnou součástí procedurálního generování. Pro vytváření náhodných čísel se využívají specializované obvody, anebo generátory pseudonáhodných čísel.

2.1.1 Lineární kongruentní generátory

Jedny z nejjednodušších generátorů pseudonáhodných čísel využívají princip lineárního kongruentního generátoru[5].

$$x_{i+1} = (a \cdot x_i + b) \bmod m \tag{2.1}$$

Kde a, b a m jsou vhodně zvolené konstanty. Takovýto generátor generuje celá čísla s rovnoměrným rozložením $0 < x_i < m$. Pro typický rozsah $\langle 0, 1)$ se musí daný výsledek dělit modulem m. Počáteční hodnota x_0 se nazývá seed. Protože generujeme náhodná čísla deterministickým způsobem, tak se časem čísla začnou opakovat. Každý generátor pseudonáhodných čísel má svoji periodicitu, po které se generovaná čísla začnou opakovat.

2.1.2 Mersenne Twister

Mersenne Twister algoritmus generuje sekvence slovních vektorů, které jsou považované za uniformní pseudonáhodná čísla v rozmezí $\langle 0, 2^w - 1\rangle$. Pomocí dělením číslem $2^w - 1$, získáme vektor čísel s realnými čísly v rozsahu $\langle 0, 1)$. $\vec{x}$ označuje slovní vektory, které jsou w-dimenzionální řádkové vektory nad dvouprvkovým polem $\mathbb{F}_2 = \{0, 1\}$, identifikující strojové slovo o velikosti w (s nejméně významným bitem vpravo). Tento algoritmus je založen na následující lineární rekurenci.

$$\vec{x}_{k+n} = \vec{x}_{k+m} \oplus (\vec{x}^{\,u}_k | \vec{x}^{\,l}_{k+1}) \cdot A \tag{2.2}$$

Kde konstanta n je stupeň rekurence, r schované v definici $\vec{x}^{\,u}_k$, $0 \leq r \leq w - 1$, číslo m, $1 \leq m \leq n$ a konstantní $w \times w$ matice A se vstupy z $\mathbb{F}_2$. Dáme generátoru $\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_{n-1}$ jako počáteční seedy. Potom generátor generuje $\vec{x}_n$ pomocí dosazení do vzorce (2.2) s $k = 0$. Dosazením za $k = 1, 2, \ldots$ generátor vygeneruje hodnoty $\vec{x}_{n+1}, \vec{x}_{n+2}, \ldots$ Na právé straně

rekurentního vztahu, $\vec{x}_k^{\,u}$ znamená horních $w-r$ bitů z $\vec{x}_k$ a $\vec{x}_k^{\,l}+1$ spodních r bitů z $\vec{x}_{k+1}$. To znamená, pokud $\vec{x}=(x_{w-1},x_{w-2},\ldots,x_0)$, tak podle definice $\vec{x}^{\,u}$ je $w-r$ bitový vektor $(x_{w-1},x_{w-2},\ldots,x_r)$ a $\vec{x}^{\,l}$ je r bitový vektor $(x_{r-1},x_{r-2},\ldots,x_0)$. $(\vec{x}_k^{\,u}|\vec{x}_{k+1}^{\,l})$ je konkatenace slov vektorů, horních $w-r$ bitů z $\vec{x}_k$ a spodních r bitu z $\vec{x}_{k+1}$. Potom je Matice A pronásobená zprava tímto vektorem. Nakonec se přičte $\vec{x}_{k+m}$, pomocí bitové operace xor a tím se vygeneruje nový vektor $\vec{x}_{k+n}$.

Pro zvýšení rychlosti generování je matice A sestavená následujícím způsobem:

$$A=\begin{pmatrix} & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \\ a_{w-1} & a_{w-2} & \cdots & \cdots & a_0 \end{pmatrix}$$

Aby se pronásobení vektoru s maticí mohlo spočítat pouze pomocí bitových operací, tak se použite tento způsob.

$$\vec{x}\cdot A=\begin{cases}\text{rightshift}(\vec{x}) & \text{if } x=0\\ \text{rightshift}(\vec{x})\oplus\vec{a} & \text{if } x=1\end{cases}$$

Kde $\vec{a}=(a_{w-1},a_{w-2},\cdots,a_0)$ a $\vec{x}=(x_{w-1},x_{w-2},\cdots,x_0)$. Také vektory $\vec{x}_k^{\,u}$ a $\vec{x}_{k+1}^{\,l}$ lze počítat pomocí bitové operace AND, díky čemuž lze rekurenci(2.2) spočítat velmi rychle pouze pomocí bitového posunu a bitových operací AND, XOR a OR.

Pro zlepšení k-distribuce na v-bitové přesnosti můžeme každé vygenerované slovo vynásobit vhodnou $w\times w$ maticí T (Tempering) zprava. Pro tempering matici $\vec{x}\mapsto\vec{z}=\vec{x}\cdot T$, jsou zvolené následující transformace:

$$\vec{y}=\vec{x}\oplus(\vec{x}>>u) \tag{2.3}$$
$$\vec{y}=\vec{y}\oplus((\vec{y}<<s)\text{ AND }\vec{b}) \tag{2.4}$$
$$\vec{y}=\vec{y}\oplus((\vec{y}<<y)\text{ AND }\vec{c}) \tag{2.5}$$
$$\vec{z}=\vec{y}\oplus(\vec{y}>>l) \tag{2.6}$$

Kde l,s,t, a u jsou čísla, $\vec{b}$ a $\vec{c}$ jsou vhodné bitové masky o velikosti slova[4].

2.2 Perlinův šum

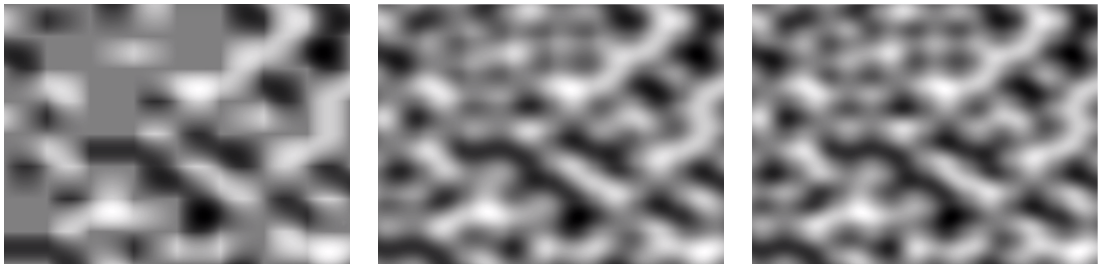
Perlinův šum je funkce pro generování pseudonáhodné hodnoty koherentního šumu, která byla vyvinutá Kenem Perlinem a publikována v SIGGRAPH roku 1985 v článku s názvem An image Synthesizer[6]. V roce 2002 byla vydaná vylepšená verze Perlinova šumu[8]. Koherentní šum znamená, že mezi jakýmikoliv dvěma body v prostoru přechází hodnoty šumu plynule. Textury využívající Perlinův šum jsou často používané za účelem zvýšení realističnosti. Možné případy použití jsou oheň, kouř, mraky.

Jeho implementace se skládá z třech kroků. Prvním krokem je sestavení n-dimezionální mřížky, kde každému bodu na této mřížce je přiřazen náhodný gradientní vektor jednotkové délky. Z důvodů optimalizace některé implementace využívají hash a lookup tabulku pro pevný počet předpočítaných gradientních vektorů. Druhým krokem je určení, do které buňky spadá zadaný bod. K němu jsou vytvořeny čtyři vzdálenostní vektory z každého bodu mřížky. Následovně je spočítán skalární vektor mezi tímto vzdálenostním a gradientním vektorem. Posledním krokem je interpolace vypočítaných hodnot.

Pouhá lineární interpolace vypadá nepřirozeně kvůli rychlým hraničním přechodům. Daleko lépe vypadá Cosinova interpolace, ale z důvodu optimalizce se zavádí Fade funkce, která v kombinaci s lineární interpolací dosahuje podobných výstupů. Původní Fade funkce(2.7) má první derivaci rovnou nule pro t=0 a t=1, díky čemuž se vyhladí artefakty způsobené hodnotou derivace funkce x. V článku z roku 2002 uvádí K. Prelin vylepšenou Fade funkci, která má i druhou derivaci rovnou nule pro t=0 i t=1. Ukázky jednotlivých interpolací můžete vidět na obrázku 2.1 a porovnání interpolačních funkcí lze vidět na obrázku 2.2.

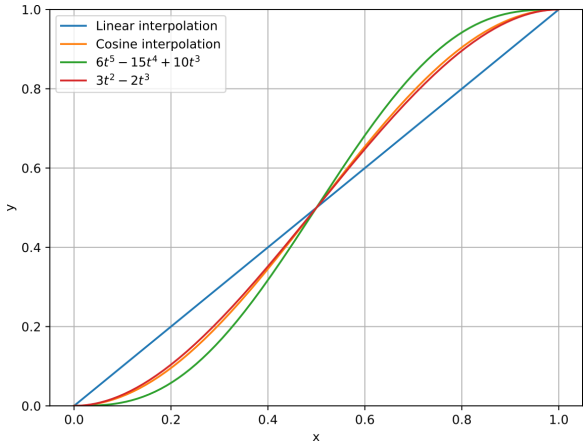
$$3t^2 - 2t^3 \tag{2.7}$$

$$6t^5 - 15t^4 + 10t^3 \tag{2.8}$$



(a) Lineární interpolace (b) Cosinova interpolace (c) Lineární + Fade

Obrázek 2.1: Porovnání interpolačních funkcí



Obrázek 2.2: Porovnání interpolací.

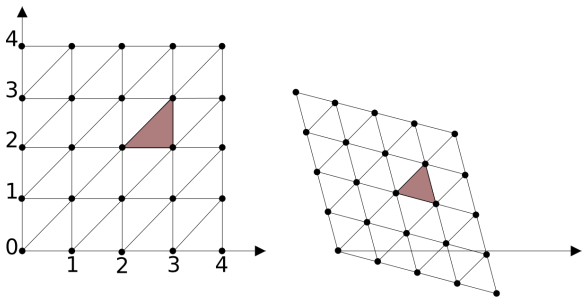
2.3 Simplexní šum

Simplexní šum je metoda pro vytvoření n-dimenzionálního šumu podobající se Perlinovu šumu. Hlavním důvodem vytvoření Simplexního šumu byla snaha odstranit nedostatky Perlinova šumu, jako jsou například: vysoká asymptotická složitost, vizuální artefakty a težká hardwarová implementace. Asymptotická složitost je u Perlinova šumu $\mathcal{O}(n2^n)$, oproti tomu složitost Simplexního šumu je $\mathcal{O}(n^2)$. Vizuální artefakty se především projevovaly pro třetí a vyšší dimenze kvůli směšovací funkci $3t^2-2t^3$. U hardwarového řešení byl největší problém

v nutnosti lookup tabulek, jenž jsou rozumné v softwarové implementaci, ale v hardwarové jsou velice drahé. Složité počítaní gradientu, kvůli kterému roste rychle asymptotická složitost. Je nutno podotknout, že tento šum byl uveřejněn v roce 2001, ještě před vylepšenou implementací Perlinova šumu v roce 2002, která některé problémy řeší[7] [3].

2.3.1 Simplexní mřížka

Pro N-dimenzionální prostorho se vybere nejjednoduší a nejkompaktnější tvar, který svým opakováním vyplní celý prostor. Pro 1D to jsou usečky o stejné velikosti, ve 2D je zřejmou volbou čtverec, ale ten má více hran než je nutné, takže nejjednoduším tvarem pro 2D je trojúhelník. Pro vyplnění celého prostoru je vhodný rovnostranný trojúhelník. Ve 3D je nejjednoduším tvarem lehce zkosený čtyřstěn. Pro výpočet zkosených koordinátů slouží vzorec (2.9). Také je zvolen pseudo-náhodný směr gradientu pro každý bod simplexního útvaru. Nejčastěji se využívá permutační tabulka nebo bitové manipulační schéma.



$$\begin{aligned} x' &= x + (x + y + \cdots) \cdot F \\ y' &= y + (x + y + \cdots) \cdot F \\ &\vdots \\ F &= \frac{\sqrt{n+1} - 1}{n} \end{aligned} \tag{2.9}$$

Obrázek 2.3: Zkosení ve 2D.

2.3.2 Interpolace

Aby se předešlo problému se sekvenční interpolací pro každou dimenzi, který vzniká u Perlinova šumu, tak Simplexní šum místo interpolace využívá jednoduchou sumaci jednotlivých podílů z každého bodu z útvaru. Každý útvar má určité pole působnosti zvolené tak, aby dosáhlo nulového příspěvku dříve než překročí hranice svého simplexního útvaru. To znamená, že bod uvnitř simplexního útvaru bude ovlivněn pouze body útvaru, do kterého spadá. Zpětné nakosení do původních koordinátů lze spočítat pomocí vzorce(2.10).

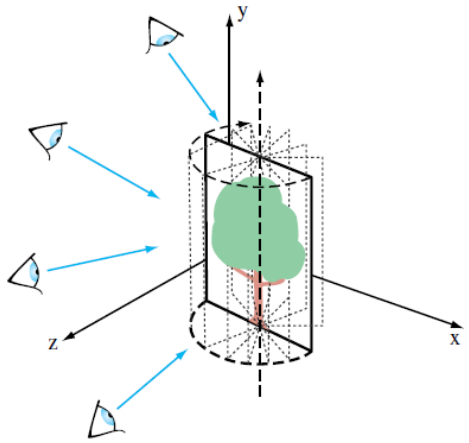
$$\begin{aligned} x &= x' + (x' + y' + \cdots) \cdot G \\ y &= y' + (x' + y' + \cdots) \cdot G \\ &\vdots \\ G &= \frac{1 - 1/\sqrt{n+1}}{n} \end{aligned} \tag{2.10}$$

2.4 Billboarding

Billboarding je v počítačové grafice technika, pomocí které se vykreslují 2D textury nejčastěji namapované na dva trojúhelníky tak, aby získaly 3D dojem. Díky tomu lze snížit počet polygonů na scéně a s ním i výpočetní výkon. Dojem 3D objektu získá pomocí natočení čtverce, na kterém je nanešena textura, směrem ke kameře.

Největší slabinou je zároveň jeho nejsilnější stránka. Při přiblížení k objektu, který je billboardován, je vidět, že je pouze nanesen na 2D plochu. Umístění je nutno přepočítávat v každém snímku, aby billboard neztratil 3D efekt a byl vždy natočen směrem ke kameře.

V dřívějších dobách, když měly počítače nízký výpočetní výkon, byly billboardy využívány k vykreslování objektů, které by kvůli svému počtu polygonů nebylo možné vykreslit. Příkladem takových objektů mohou být nepřátelé nebo stromy. V dnešní době jsou využívány k vykreslování mraků, ohně nebo v kombinaci s geometry shaderem na vykreslování spousty malých objektů, jako jsou třeba sněhové vločky, kapky deště a mnohé další.

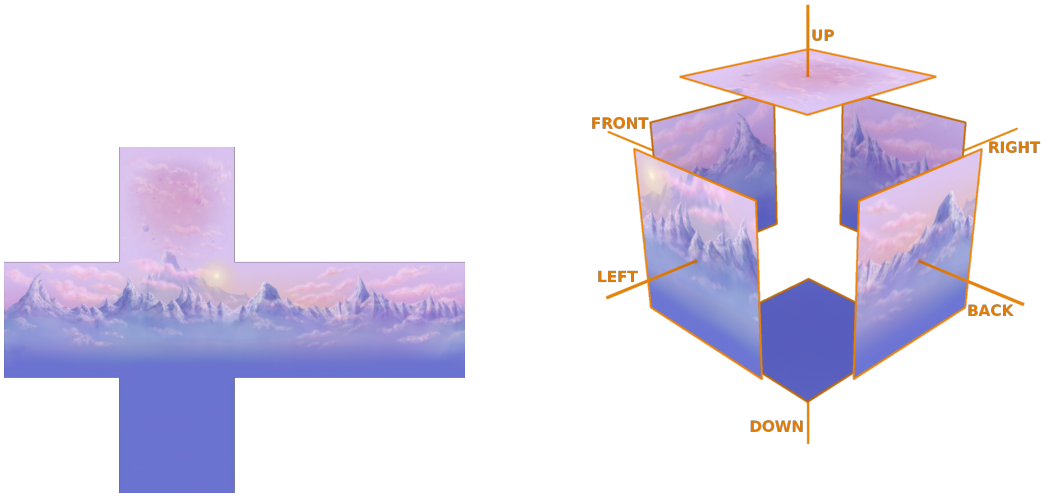


Obrázek 2.4: Ilustrační obrázek billboardingu[1].

2.5 Skybox

Skybox je technika využívaná pro vytvoření okolní prostředí scény, nejčastěji oblohy, mraků, hor, oceánů či velkých budov v pozadí. Všechny nedosažitelné objekty v pozadí jsou nanesené na textury, které jsou promítnuty pomocí cube mappingu. Kamera je umístěna do středu skyboxu, takže se nikdy nedostane k horizontu scény. Tato technika dělá scénu daleko větší než doopravdy je. A proto tato technika nebo její modernější ekvivalent skydome je využita téměř v každé hře. Skydome je pricipiálně stejný jako skybox, ale místo logického mapovaní textur na krychly se využívá polokoule. Pro správné vykreslení skyboxu je nutné jej vykreslit jako první na scénu s vypnutým depth testem a zbytek scény jej překryje. Příklad skyboxu je na obrázku 2.5.

[1]Převzato z http://ddstosic.50webs.com/dwarf.htm
[2]Převzato z https://gamebanana.com/textures/2548



(a) Rozbalený skybox (b) Složený skybox

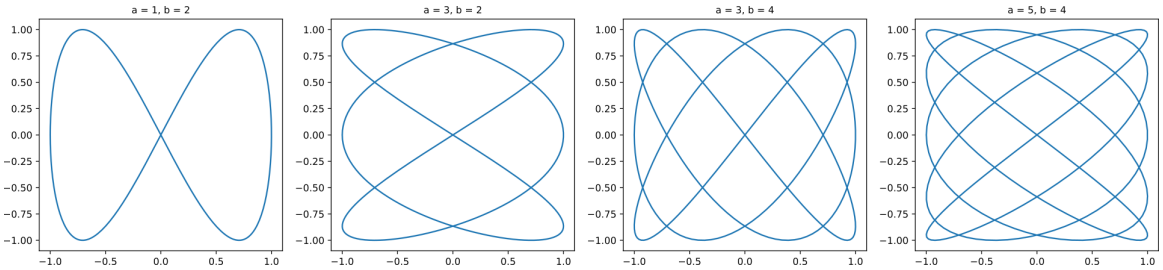
Obrázek 2.5: Příklad skyboxu[2].

2.6 Lissajousovy obrazce

Lissajousovy obrazce jsou grafem soustavy paremetrických rovnic 2.11.

$$x = A\sin(at + \delta) + C, \qquad y = B\sin(bt) + D \tag{2.11}$$

Kde A, B ovlivňují šířku a výšku vykreslované obrazce. Parametry a, b, především jejich poměr $\frac{a}{b}$, má vliv na výsledný tvar daného obrazce a proměnné C, D zařizují posun po osách x a y. Parametr δ se stará o vzájemný posun křivek vůči sobě. Tyto křivky se používají pro popis komplexních harmonických pohybů. Dají se využít i pro iluzi nedeterministického pohybu, jako je například skupina malých objektů, kde se každému objektu náhodně posune fáze. Tato vlastnost může být použita například pro simulaci pohybu roje včel. Příklady obrazců jsou na obrázku 2.6.

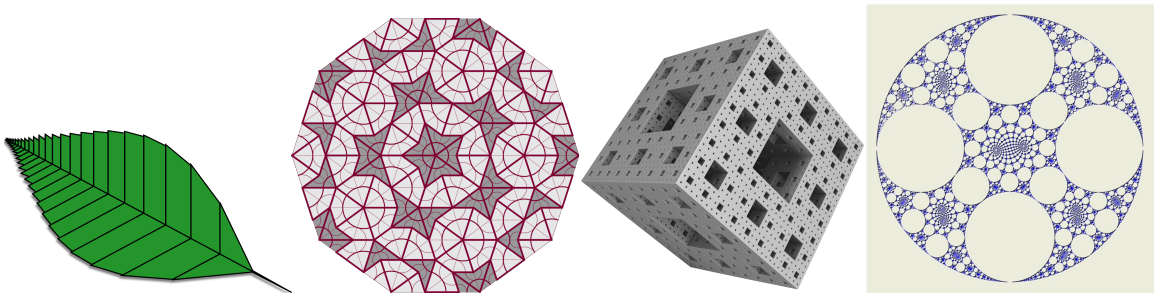


Obrázek 2.6: Lissajousovy obrazce.

2.7 Fraktály

Fraktály byly poprvé zkoumány matematikem Benoit Mandelbrot v roce 1975. Jejich název pochazí z latinského slova fractus, což znamená nepravidelné nebo roztříštěné. Jednou

z hlavních charateristik fraktálu je soběpodobnost, ať pozorujeme útvar v jakémkoliv měřítku, tak se stále opakuje určitý vzor. Často mívají na první pohled velmi složitý tvar, ale jsou generovány pomocí opakováním velmi jednoduchých pravidel. Nejčastějšími způsoby pro generování fraktálu jsou iterativní funkční předpisy (IFS), L-systémy, atraktory nebo konečné dělicí pravidla. Příklady na jednotlivé způsoby naleznete na obrázku 2.7. V počítačové grafice jsou využívány nejrůznějšími způsoby. Pomocí fraktálu se dá generovat terén, systém řek, vodní vlny a samozřejmě textury[1] [2].



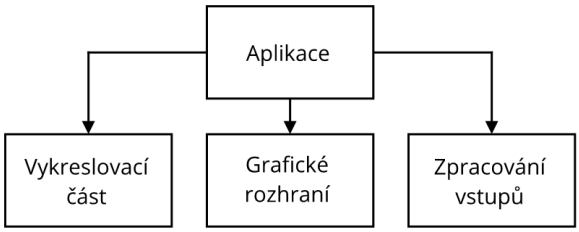
(a) L-systém (b) Dělící pravidla (c) IFS (d) Atraktor

Obrázek 2.7: Příklady fraktalů.

Kapitola 3

Návrh aplikace

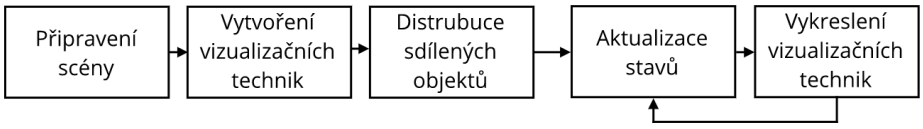
Samotná demonstrační aplikace je z návrhového hlediska rozdělena na tři logické části, které lze vidět na obrázku 3.1. Ve své demonstrační aplikaci jsem se rozhodl udělat krátkou animaci, dvou bojujících lodí s využitím veškerých vytvořených efektů. Dále bude postupně rozebrán návrh vykreslovací smyčky společně s využitými částmi GPUEnginu a následně jednotlivé efekty.



Obrázek 3.1: Návrh aplikace.

3.1 Vykreslovací část

Vykreslovací část se skládá ze dvou dílů: inicializační části a vykreslovací smyčky. V inicializační části probíhá nastavení scény, vytvoření všech vizualizačních technik a rozdistribuování sdílených objektů. Ve vykreslovací smyčce, která probíhá každý snímek, se aktualizují stavy všech objektů a následně se provádí vykreslování pomocí vizualizačních technik. Diagram vykreslovací části lze naleznout na obrázku 3.2.



Obrázek 3.2: Vykreslovací část.

3.2 Vizualizační techniky

Vizualizační technika je rozhraní, které odděluje obecný popis objektu a jeho OpenGL reprezentaci. Je rozdělena na dvě části: aktualizační a vykreslovací. V aktualizační části

probíhají výpočty nutné pro vykreslování, což může být zpracování nových objektů a jejich převod na OpenGL reprezentaci anebo přepočítávání koordinátů pro správné zobrazování billboardů. Vykreslovací část sváže vytvořené datové struktury s grafickou kartou a vykreslí daný objekt. Díky tomuto rozhraní je velice jednoduché si udržet oddělený popis objektu, který je vhodný pro animace nebo počítaní průniků se štíty a OpenGL reprezentaci.

3.3 Příprava scény

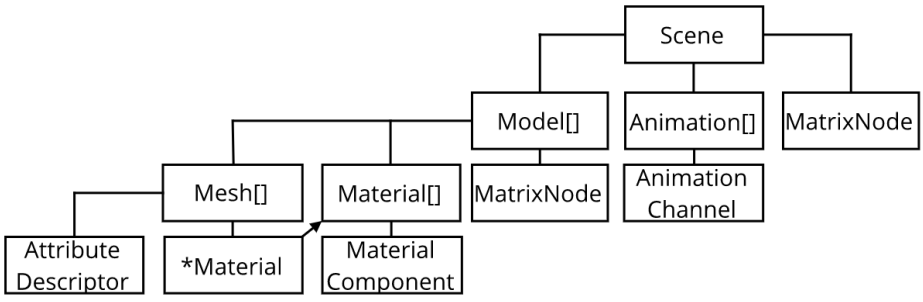
V první části je nutné nahrát přípravené modely s texturami do vhodné reprezentace pro zpracování GPUEnginem. Aby bylo možné pracovat s modely, musí být aplikace schopná načíst daný model ze souboru a převést jej do vhodných datových struktur pro GPUEngine. Ten pro scénu nabízí datovou strukturu, kterou můžete nalézt na obrázku 3.3.

Samostatná scéna se skládá z animací a modelů. Model je poté rozdělen na materiály, což mohou být normálové mapy, textury či difúzní barva a mnohé další. Pokud by si chtěl uživatel sám naspecifikovat materiál, tak GPUEngine nabízí rozhraní pro základní materiály, u kterých existuje možnost zaregistrovat novou sémantiku vložených dat.

Druhou částí modelu jsou meshe, jedná se o skupiny vrcholů, které jsou svázány pomocí ukazetelů s naspecifikovanými materiály ve scéně. Tyto skupinky vrcholů mohou být složeny z více atributů. V jednotlivých atributech se může nacházet například pozice a texturovací koordináty. Protože se jedná o obecné uložiště dat, tak jsou informace o datech předávány v attribute descriptorech, které popisují uložená data, specifikují velikost dat, použitý datový typ, offset a stride pro jednotlivé atributy.

Pro zpracování co nejvíce formátů byla využita knihovna Assimp, pro kterou je v GPUEnginu vytvořený addon `AssimpModelLoader`. Ten je schopen převést namodelovanou scénu ze souboru do datové struktury připravené GPUEnginem. Avšak tato knihovna je schopná nahrát pouze model a informace o texturách, ale samostatné obrázky musí být nahrány pomocí jiné knihovny. Pro nahrávání obrázků byla využita již použitá knihovna Qt pro grafické rozhraní, která má pro tato nahrávání v GPUEnginu addon `QtImageLoader`. Posledním krokem je převod načteného obrázku na OpenGL texturu.

V této části se také nastaví všechny animace pro celou scénu, které jsou následnově spuštěny.

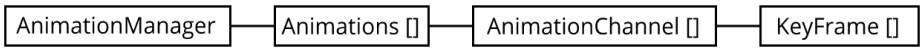


Obrázek 3.3: Reprezentace scény GPUEngine.

3.4 Animace

Animace mají základní podporu v GPUEnginu v podobě správce animací, který si v sobě ukládá jednotlivé animace, skládající se z jednotlivých klíčových snímků. Diagram tříd

z GPUEnginu můžete vidět na obrázku 3.4. Animace se skládá z animačních kanálů, které udávají počet vlastností, jenž se budou měnit v průběhu času. Pro základní typ animace, který využívá transformační matice, existuje v GPUEnginu již nadefinovaný MovementAnimationChannel. Avšak pro jakoukoliv jinou animaci, která nevyužívá transformační matici, musí být napsána nová třída implementující AnimationChannel rozhraní. Samotný animační kanál se skládá z jednotlivých klíčových snímků. Klíčový snímek je určité místo v čase, na kterém se definuje hodnota pro specifickou vlastnost objektu. Touto vlastností může být například velikost, barva či pozice. Pomocí interpolace jednotlivých vlastností dosáhneme kýženou animaci.



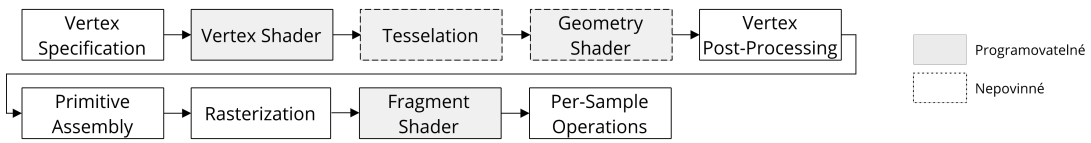
Obrázek 3.4: Diagram animační třídy GPUEngine.

Interpolace probíhá pomocí nadefinovaného interpolátoru. GPUEngine nabízí tři základní interpolátory: Lineární, Sférický lineární a výběr bližšího snímku. Při interpolaci pomocí výběru bližšího snímku se rozhodne, který z klíčových snímků je blíže a ten nastaví jako aktuální. Pokud by byla potřeba jiná interpolace, tak může být naimplementována pomocí zdědění šablonovaného rozhraní KeyframeInterpolator.

V rámci své práce jsem potřeboval animovat nejen pomocí transformačních matic, pro které má již engine připravenou podporu. Některé animace probíhají pouze interpolací jednoduchých datových typů jako jsou celá či desetinná čísla nebo vektorů. Rozhodl jsem se vytvořit speciální šablonovanou třídu, která dokáže vytvořit animační kanál nad základními typy jako int, float, vektory a dalšími pomocí lineárního interpolátoru. Základní správce animací neposkytoval potřebnou funkcionalitu, pro mazání animací před jejich dokončením, z tohoto důvodu byl rozšířen o potřebnou funkcionalitu.

3.5 Programy pro vizualizační techniky

V druhé fázi je potřeba vytvoření programovatelných bloků v OpenGL pipeline. Tyto programovatelné bloky, které se programují pomocí jazyka GLSL (OpenGL Shading Language), se nazývají shadery. Programovací jazyk GLSL je založen na programovacím jazyku C, ale definuje vlastní datové typy a vestavěné funkce pro usnadnění práce s vykreslováním. OpenGL nabízí 5 programovatelných bloků ve vykreslovací smyčce: Vertex, Tesselation Control, Tesselation Evalution, Geometry a Fragment shader. Existuje ještě jeden programovatelný blok nazývaný Compute shader, který není určený pro vykreslování, nýbrž pro počítání libovolných úloh na grafické kartě. Aby mohla být vykreslovací pipeline považována za úplnou, je potřeba naimplementovat minimálně Vertex a Fragment shader. Celou vykreslovácí pipeline můžete vidět na obrázku 3.5.



Obrázek 3.5: OpenGL pipeline.

Na vytvoření shaderu GPUEngine nabízí třídu Shader, která je schopná přeložit GLSL shader. Následně třídou Program je schopen slinkovat shadery do programu pro grafickou kartu. V práci byly využity všechny druhy shaderu kromě teselačních.

3.6 Nárazy střel

Pro nárazy střel do štítů je potřeba spočítat místa střetu těchto objektu. Pro výpočet průniků nabízí GPUEngine pouze 3 základní třídy, které jsou schopné zjistit průnik a to vždy mezi polopřímkou a jedním ze tří objektů: koule, trojúhelník či mesh. Aby bylo možné spočítat střety mezi štíty a lasery, anebo mezi štíty a energetickými střelami, je potřeba zjistit průnik mezi geomterickými obrazci jako jsou úsečka a koule nebo mezi dvěma koulemi. Pro výpočet střetu mezi úsečkou a koulí byla využita třída z GPUEnginu, která dokáže vypočítat průnik mezi polopřímkou a koulí pomocí vzorce 3.1.

$$D_r^2 + 2 \cdot [D_r \cdot (O_r - O_s)] + (O_r - O_s)^2 - R_s^2 = 0 \tag{3.1}$$

Kde D_r je vektor určující směr a O_r počáteční bod polopřímky. Parametr O_s značí střed koule a R_s její poloměr. Výsledkem průniku je parametr t z parametrického vyjádření přímky ($P = p + at$).

Pro zjištění průniku úsečky a koule, musí být parameter t menší, než je velikost úsečky, aby došlo k průniku. K dopočítání bodu průniku pak stačí k počátečnímu bodu úsečky přičíst t násobek normalizovaného směrového vektoru.

Druhým případem, jenž může nastat, je střet energetické střely se štítem, u kterého zjišťujeme, zda došlo ke střetu dvou koulí. Pro tento úkon se využívá nerovnice 3.2. Pokud dojde ke střetu, tak se není potřeba dopočítávat ke všem bodům, které vzniknou průnikem dvou koulí, nýbrž jen k jednomu, který určuje, v jakém místě ke střetu došlo. K tomuto místu se lze dopočítat pomocí rovnice 3.3. Kde S_{1o} jsou koordináty středu první koule, $\hat{S_d}$ je normalizovaný vektor vzdálenosti mezi koulemi ($S_d = S_{2o} - S_{1o}$) a S_{1r} je poloměr první koule.

$$S_{1o}^2 + S_{2o}^2 < S_{1r} + S_{2r} \tag{3.2}$$

$$S_{1o} + \hat{S_d} \cdot S_{1r} \tag{3.3}$$

3.7 Grafické efekty

Efekty jsou z hlediska návrhu rozděleny na tři části. Tou první je obecný popis objektu spolu se základní třídou pro geometrii. Druhou část představuje manažer, který se stará o všechny instance daného efektu a obstarává základní správu efektů. Poslední částí je výše zmíněná vizualizační technika, která převede objekt na vhodnou OpenGL reprezentaci a udržuje si tyto objekty připravené pro vykreslení. Dále postupně popíši navrhnuté efekty, kterými jsou štít, laser, energetické střely a elektrický oblouk.

3.7.1 Generování geometrie koule

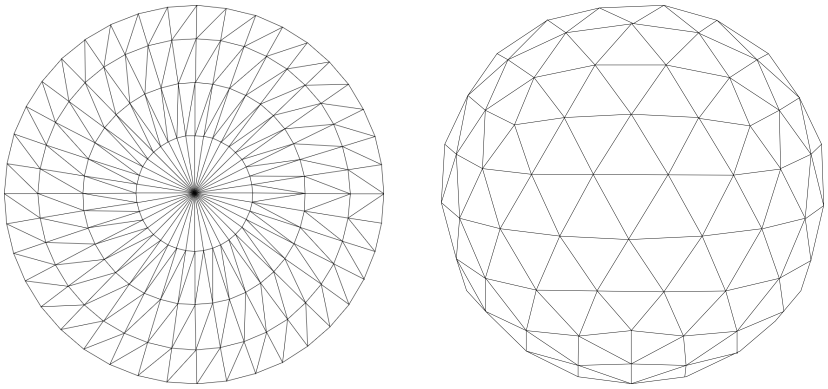
Protože dva z navrhnutých efektů jsou geometricky reprezentovány pomocí koule a v počítačové grafice jsou základními primitivy bod, úsečka, trojúhelník či čtyřúhelník, bude v této kapitole popsán způsob vytvoření koule pomocí těchto primitiv.

Pro generování koule jsou dva nejčastější způsoby. Prvním je vytvoření jednotkového icosahedronu (pravidelný dvaceti stěn), který se bude nadále plynule rozdělovat a zaoblovat. Druhý způsob se zakládá na využití goniometrických funkcí pro vygenerování bodu na kouli. V demonstrační aplikaci je pro generování koule využito právě goniometrických funkcí, proto bude tento způsob dále rozebrán podrobněji.

Při vytváření koule pomocí goniometrických funkcí je potřeba aplikovat vzorce 3.4.

$$\begin{aligned}(X_1, Y_1, Z_1) &\rightarrow (\cos(a) \cdot \sin(b), \sin(a) \cdot \sin(b), cos(b)) \\ (X_2, Y_2, Z_2) &\rightarrow (\cos(a) \cdot \sin(b + \varphi), \sin(a) \cdot \sin(b + \varphi), \cos(a + \varphi))\end{aligned} \tag{3.4}$$

Kde a a b označují rotace kolem jednotlivých os. Parametr φ označuje, jak daleko budou jednotlivé segmenty od sebe. Pro vygenerování celé koule je potřeba projít rotačním parametrem a interval $\langle 0, \pi \rangle$ a parametrem b $\langle 0, 2\pi \rangle$. Aby nedošlo k vytvoření artefaktů skládajících se segmentů přes sebe, je potřeba procházet tyto intervaly se skokovou změnou, kterou značí proměnná φ. Na obrázku 3.6 můžete vidět pro srovnání oba způsoby výstupní geometrie.



(a) Goniometrické funkce (b) Dělení icosahedronu

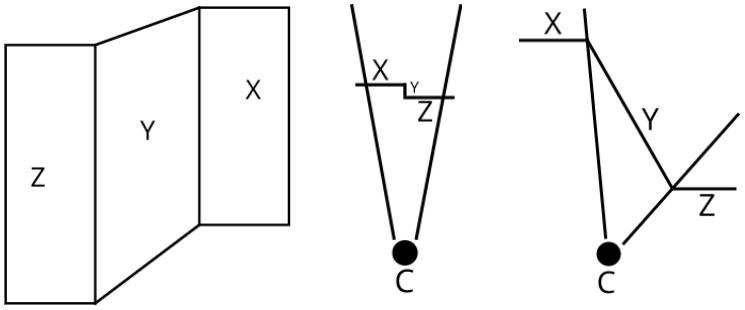
Obrázek 3.6: Srovnání geomterií.

3.7.2 Střely

Přestože každá střela může mít svou unikátní geometrii, je dosaženo jednotného přístupu pro správu střel, díky základní šablonované třídě, která obsahuje rozhraní, jenž musí splňovat každá střela, spolu se základními funkcemi. Pro každou střelu je nutno naimplmentovat metody pro vytvoření animací. První animací je trajektorie letu střely a druhou je zánik střely při nárazu do štítu.

3.7.3 Lasery

Lasery pro svoji specifikaci potřebují počáteční bod, koncový bod, šířku a barvu. Laser je reprezentován pomocí tří billboardů 2.4 ve tvaru lichoběžníku. Prostřední lichoběžník je natáčen podle pozice a orientace laseru. Boční billboardy jsou sestaveny tak, aby byly stále natočeny směrem na kameru a při čelním pohledu vytvořily kruhový tvar. Ilustrační obrázky této techniky jsou znázorněny na obrázku 3.7.
Kde X, Y, Z jsou jednotlivé billboardy a C reprezentuje kameru.



(a) Billboardy (b) Přímý pohled (c) Pohled z boku

Obrázek 3.7: Ilustrační obrazky billboardů.

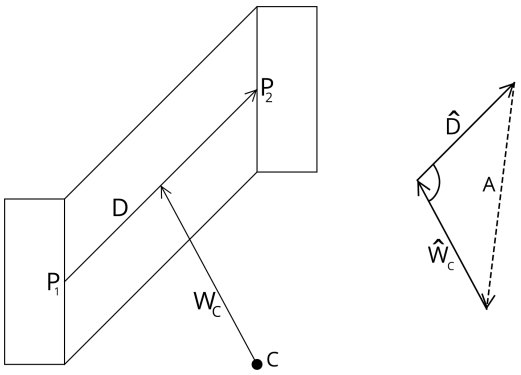
Tvorba laseru

Klíčem k vytvoření laseru je převedení jeho 3D světových koordinátů na koordináty okna. Z důvodu převedení laseru z světových koordinátu, je potřeba vyřešit, jak se jednotlivé billboardy budou zmenšovat vlivem vzdálenosti od kamery. Tento výpočet může být proveden za použití jednoduchého vzorce 3.5.

$$\frac{DS_F}{|P - C|} \tag{3.5}$$

Kde DS_F značí parametr určující, s jakou rychlostí se budou billboardy zmenšovat s rostoucí vzdáleností. P je počáteční nebo koncový bod laseru a C označuje pozici kamery v prostoru.

Abychom byli schopni laser smršťovat a rozpínat vzhledem k velikosti úhlu mezi kamerou a laserem, je potřeba tento úhel zjistit. K tomuto úhlu se lze dopočítat vzorcem 3.6.



Obrázek 3.8: Grafická reprezentace vzorce 3.6.

$$\begin{aligned} A &= |\hat{D} \cdot \hat{W}_c|^S \\ W_c &= C - \frac{P_1 + P_2}{2} \\ D &= P_2 - P_1 \end{aligned} \tag{3.6}$$

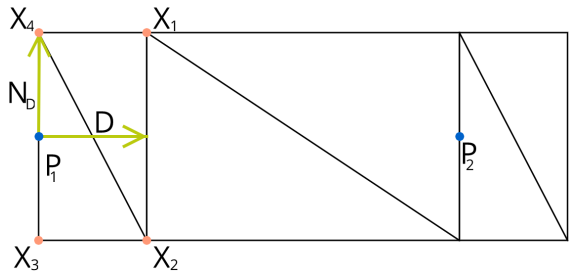
Kde $\hat{D}$ značí normalizovaný směrový vektor, P_1 počáteční a P_2 koncový bod laseru ve světových koordinátech. $\hat{W}_c$ je normalizovaný směrový vektor pohledu do středu laseru. Aby bylo možné zajistit specifickou rychlost smršťování a rozpínaní, tak se výsledný úhel, který je v rozpětí $\langle 0, 1 \rangle$, umocňuje parametrem S ovlivňujícím tuto rychlost.

Z důvodu rozpínání či smršťování, je potřeba upravit vzdálenostní faktor aktuálně vzdálenějšího bodu vzhledem ke kameře vzorcem 3.7.

$$D_{ff} = D_{ff} + (D_{fn} - D_{ff}) \cdot A \tag{3.7}$$

Kde D_{ff} je vzdálenostní faktor vzdálenějšího a D_{fn} bližšího bodu, který je výsledkem rovnice 3.5 a A je úhel mezi kamerou a středem laseru z rovnice 3.6.

Pro konečnou pozici bodů billboardu, je potřeba zjistit normalizovaný vektor směru laseru a jeho normálový vektor. Ten je potřeba vynásobit šířkou laseru společně se vzdálenostím faktorem, který určuje perspektivní zkreslení. Výpočet prvních čtyř bodů společně s obrázkem 3.9, lze nalézt níže.



$$
\begin{aligned}
X_1 &= (P_{1x} - N_x + D_x \quad , P_{1x} - N_y + D_y) \\
X_2 &= (P_{1x} + N_x + D_x \quad , P_{1x} + N_y + D_y) \\
X_3 &= (P_{1x} - N_x \quad , P_{1x} - N_y) \\
X_4 &= (P_{1x} + N_x \quad , P_{1x} + N_y)
\end{aligned}
$$

Obrázek 3.9: Vytvoření geometrie.

Z důvodu lichobežníkového tvaru billboardů nelze použít UV mapování pro textury. Když je při vykreslení prováděn rozklad na objektu na jednotlivé trojúhleníky, jedna z diagonál je vybrána jako hrana. Pokud tato hrana při namapování prochází přímo středem textury, tak není problém použít UV koordináty, pro čtvercovou texturu. Avšak v případě lichoběžníku hrana nikdy nebude procházet středem. Při použití klasických UV koordinátů pro namapování čtvercové textury na lichoběžník dochází ke špatné interpolaci z důvodu rozdělení na trojúhelníky. Výsledný efekt můžeme vidět na obrázku 3.10. Proto je potřeba, místo klasických UV mapovacích koordinátů, použít projekční texturovací koordináty. Výpočet pro vizualizaci laseru byl převzat z demonstračního příkladu na zmíněnou techniku[1].

(a) Textura (b) UV mapování (c) Projektivní mapování

Obrázek 3.10: Texturování lichoběžníku.

3.7.4 Energetické střely

Energetické střely jsou reprezentovány pomocí koule, u které se specifikují pouze pozice, poloměr a směr střely. Při nárazu se štítem se střely začnou rychle zmenšovat, pro vytvoření efektu pohlcení štítem.

[1] https://gitlab.com/geoff-nagy/laser-demo-simple/

Tvorba střely

Pro sestavení střely je potřeba vytvořit kouli o poloměru 1, která se bude zvětšovat a posunovat pomocí modelové matice. Pro vytvoření geometrie koule viz. 3.7.1.

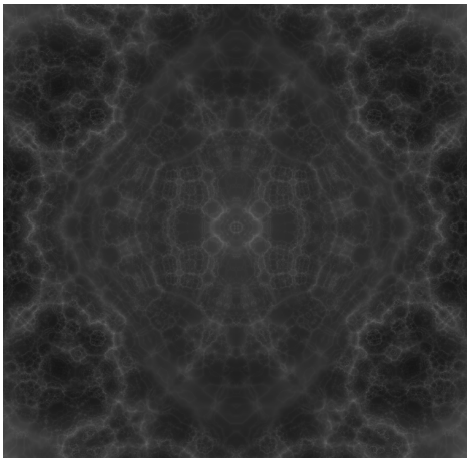
Aby se předešlo texturování pomocí UV souřadnic, pro které by se muselo vytvořit korektní namapování geometrie koule na texturu. Byla využita procedurálně generovaná textura, vytvořená pomocí 3D Perlinova šumu 2.2, jehož vstupem je pozice. Následně je tento šum obarven.

3.7.5 Štíty

Štíty si kromě svého popisu, jako je poloměr nebo středový bod, ukládají navíc informace po zásahu laserem nebo energetickou střelou. Při nárazu se vygenerují efekty propalující se střely skrz štít a pulzní vlna, které zviditelní štít pomocí dočasného zesvětlení.

Tvorba štítu

Štít je tvořen pomocí koule, jejíž tvorba je popsána v kapitole 3.7.1. Pro výsledný efekt štítu je použita procedurální textura využívající fraktálu 2.7. Fraktál je zadaný pomocí systému iterovaných funkcí 3.8. Aby štít nevypadal jako statický objekt, je pomocí Lissajousového obrazce 2.6 posouvána pozice ve fraktálu, čímž vytváří efekt animace.



$$\begin{aligned} x &= abs(x) \\ y &= abs(y) \\ m &= x^2 + y^2 \\ x &= x/m - cx \\ y &= y/m - cy \end{aligned} \tag{3.8}$$

Obrázek 3.11: Vizualizace fraktálu 3.8.

Pulzní vlna

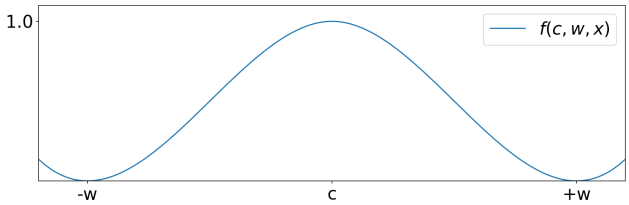
Pulzní vlnu specifikuje bod nárazu a parametr t z rozsahu $\langle 0, 1 \rangle$, který určuje, v jaké vzdálenosti se nachází pulzní vlna od bodu nárazu. Pro výpočet je potřeba určit sférickou vzdálenost aktuálně počítaného bodu a místem nárazu, dle vzorce 3.9.

$$\begin{gathered} r \cdot \arctan(length(P_1 \bullet P_2), P_1 \times P_2) \\ length(x) = \sqrt{x \cdot x} \end{gathered} \tag{3.9}$$

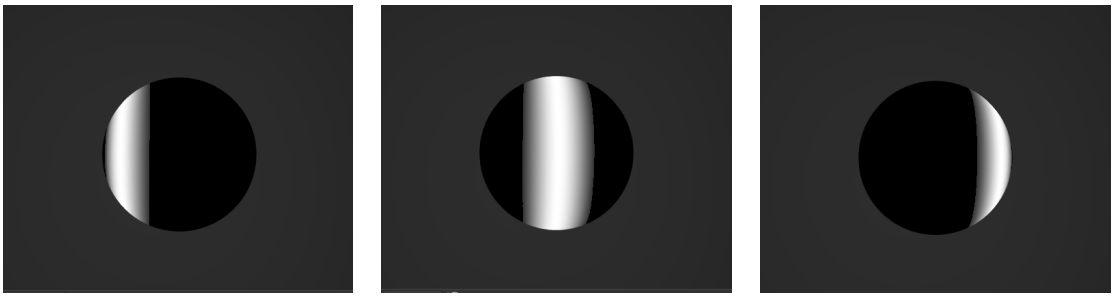
Kde r je poloměr štítu, P_1 a P_2 jsou body, pro které chceme počítat sférickou vzdálenost. Pro správné výsledky se body musí posunout zpět k počátečnímu bodu soustavy pomocí inverzních modelových matic. Pro výsledný gradientní pás, je potřeba upravit parametr t

tak, aby svým nejvyšším číslem v intervalu odpovídal sférické vzdálenosti dvou nejvzdálenějších bodů na štítu. Zároveň požadujeme, aby tento prstenec zmizel na druhé straně štítu. Proto je potřeba konečný výsledek vynásobit konstantou x, která je závislá na šířce gradientního pásu a zajistí postupné zmizení. Vzorec pro úpravu parametru t je následující: $t = t \cdot x \cdot r \cdot \pi$. Jakmile máme upravený parametr t a známe sférickou vzdálenost bodů, můžeme ve vzdálenosti, kde je $t > d \wedge t < d + w$, kde d je sférická vzdálenost a w je šířka gradientního pásu, vytvořit interpolací pro gradientní přechod. Pro interpolaci hodnot na gradientním pásu byla použita funkce 3.10 s hodnotami $c = 0.5, w = 0.5$, jejíž graf lze vidět na obrázku 3.12. Výsledný prstenec naleznete na obrázku 3.13.

$$f(c, w, x) = 1.0 - u^2 \cdot (3.0 - 2.0 \cdot u), u = |\frac{x - c}{w}| \tag{3.10}$$



Obrázek 3.12: Kubický puls.



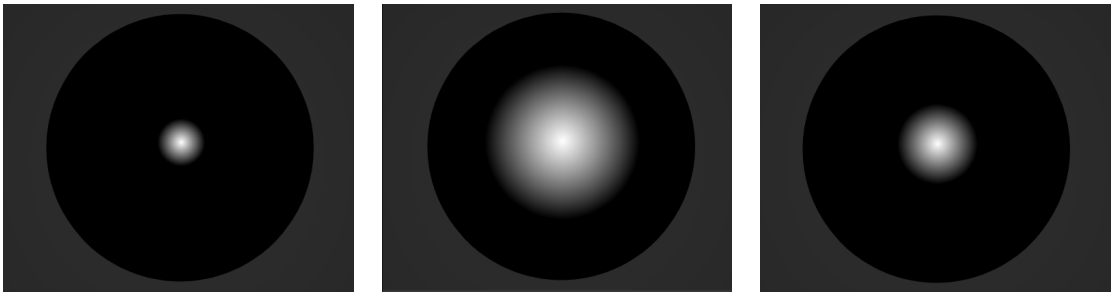
(a) t = 0.30 (b) t = 0.50 (c) t = 0.70

Obrázek 3.13: Gradientní přechod pulzní vlny.

Efekt propalující se střely

Pro efekt propalující se střely štítem vzniká, v místě nárazu, lineární gradientní přechod, jehož velikost se mění v průběhu času a maximální hodnoty dosáhne v polovině své doby životnosti. Velikost gradientu je popsána rovnicí 3.10, s parametry $c = 0.5, w = 0.5, x = t$, kde t je normalizovaná doba životnosti efektu. Výsledek této rovnice je vynásoben polovinou poloměru štítu, aby efekt zohlednil velikost štítu.

Abychom dostali lineární interpolaci při měnící se velikosti, je potřeba vypočítanou sférickou vzdálenost podělit aktuální velikostí gradientu. Výsledný efekt lze vidět na obrázku 3.14.



(a) t = 0.15 (b) t = 0.40 (c) t = 0.80

Obrázek 3.14: Gradientní přechod při nárazu do štítu.

3.7.6 Elektrický oblouk

Elektrický oblouk je složen z několika úseček, které na sebe navzájem navazují. Tyto úsečky jsou převedeny na billboardy 2.4, o zadané šířce, pro které se musí dopočítat vzájemné navazování. Následně je pro ně vytvořena textura pomocí kombinace Perlinova 2.2 a Simplexního šumu 2.3.

Tvorba geometrie elektrického oblouku

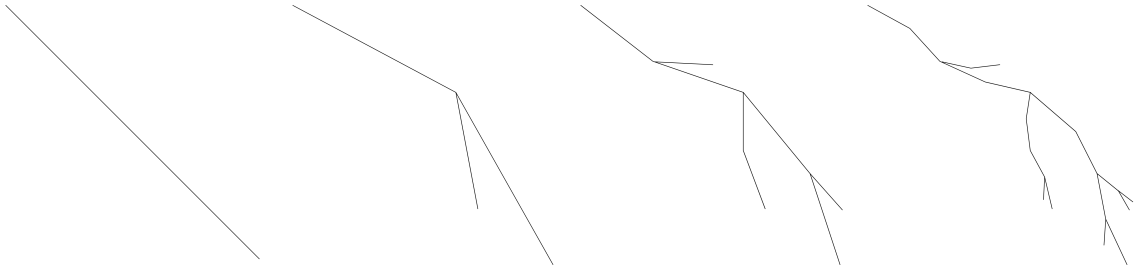
Algoritmus pro vytváření geometrie, lze využít k vytvoření zakřivení geomterie elektrického oblouku. Avšak pokud tomuto algoritmu povolíme následné štěpení objektů, výsledná geomterie připomíná blesk. Prvním krokem algoritmu je rozdělení úsečky podle posunutého středového bodu. Pro posunutí středového bodu je potřeba zjistit normalizovaný normálový vektor úsečky, který je vynásoben pseudonáhodným číslem v rozsahu $\langle -\varphi, \varphi \rangle$, kde φ je zadaný maximální posun. Tento vektor určuje nově vzniklý středový bod, podle kterého bude tato úsečka rozdělena na dvě.

Pokud bychom chtěli získat geometrii blesku, vytvoříme náhodné číslo z rozsahu $\langle 0, \frac{1}{d} \rangle$, kde d značí počet hloubku zanoření ve větvení. Velikost tohoto čísla nám určí, zda se má vytvářet nové větvení na tomto segmentu. Jediným krokem pro vytváření nové větve, je zjistění vektoru z počátečního po středový bod. Tento vektor je pootočen v rozsahu $\langle -\alpha, \alpha \rangle$, kde α určuje maximalní úhel odklonu nového segmentu. Nová větev je poté vytvořena od středového bodu po nově vzniklý vektor přičtený ke středovému bodu a je mu dána vyšší hodnota hloubky ve větvení. S každou iterací algoritmu se offset zmenšuje na polovinu. Příklad jednotlivých iterací algoritmu můžete vidět na obrázku 3.15.

Problém navazování segmentů

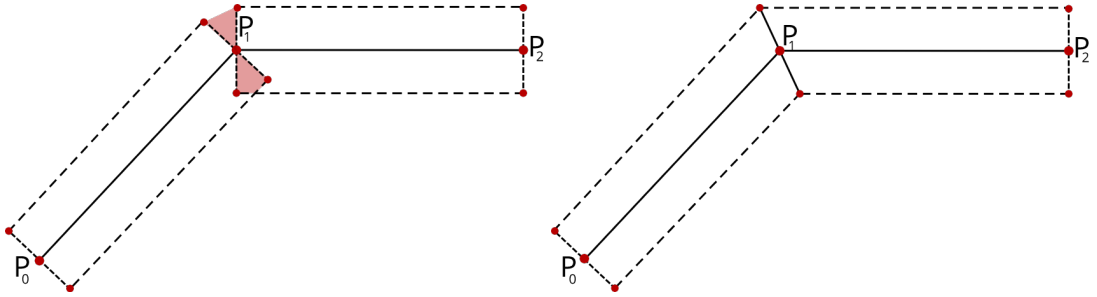
Elektricky oblouk se skládá z úseček, které na sebe navazují. Při převodu na billboardy o zadané šířce vzniká problém s navazováním jednotlivých billboardů. Ilustrační obrázek problému 3.16.

Jak lze vidět na obrázku 3.16, je potřeba dopočítat 6 bodů pro získání korektní návaznosti. První čtyři body lze získat pomocí normalizací normály směrových vektorů, které se vynásobí potřebnou šířkou. Pro prostřední body se musí sečíst směrové vektory obou úseček a následně tento vektor normalizovat a otočit o 90°. Tím získáme vektor, na kterém se nachazí hledané body. Pro dopočítání správné šířky je potřeba provést projekci tohoto vektoru na jednu z normál pomocí skalárního součinu, čímž získame všechny informace



(a) 1 iterace (b) 2 iterace (c) 3 iterace (d) 4 iterace

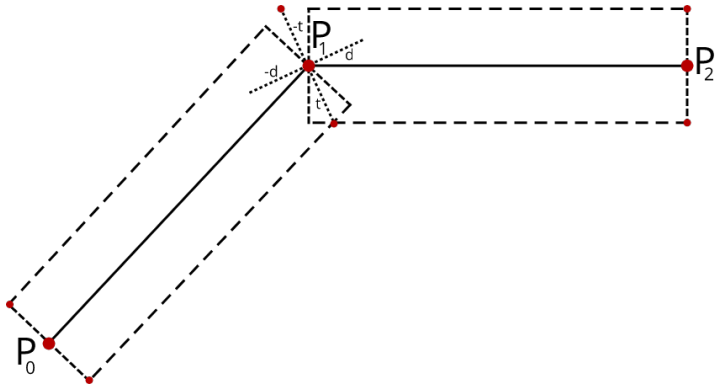
Obrázek 3.15: Iterační generování geomterie.



(a) Špatná návaznost (b) Korektní návaznost

Obrázek 3.16: Ilustrační obrázek problému návaznosti segmentů.

potřebné pro nalezení posledních dvou bodů. Grafickou reprezentaci společně s rovnicemi pro výpočet jednotlivých bodů 3.11 lze nalézt na obrázku 3.17.



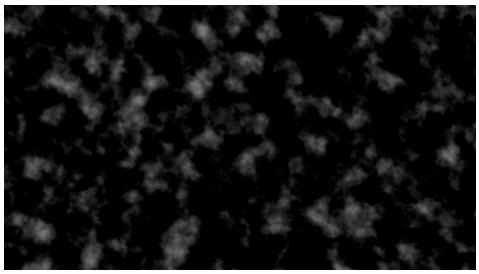
$$
\begin{aligned}
l_1 &= P_1 - P_0 \\
l_2 &= P_2 - P_1 \\
d_f &= P_2 - P_0 \\
d &= \hat{d}_f \\
t &= n_d \\
\Delta_c &= \Delta/(t \bullet n_1) \\
p_{0_a}, p_{0_b} &= P_0 \pm \hat{n}_1 \cdot \Delta \\
p_{1_a}, p_{1_b} &= P_1 \pm t \cdot \Delta_c \\
p_{2_a}, p_{2_b} &= P_2 \pm \hat{n}_2 \cdot \Delta
\end{aligned}
\tag{3.11}
$$

Obrázek 3.17: Grafická reprezentace vektorů z rovnice 3.11.

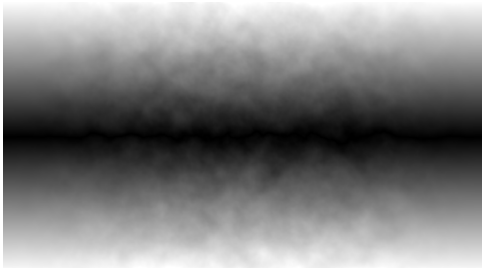
Tvorba textury výboje

Pro vytvoření efektu blesku je zapotřebí součet kombinací 3D simplexních šumu 2.3 s různou amplitudou a frekvencí. Vstupem složeného 3D simplexního šumu jsou na dvou koordinátech zmenšené UV souřadnice a čas, který vytvoří animaci. Tento šum se skládá ze 4

simplexních šumů s váhami: $0.53, 0.26, 0.13, 0.06$, přičemž se při vytváření každého z těchto šumů posílá $2\times$ větší frekvence, než tomu předešlému. Tyto váhy jsou zvoleny tak, abychom dostali požadovaný výsledný tvar a zároveň nechali rozsah hodnot stále v rozmezí $\langle -1, 1\rangle$. Příklad vygenerovaného šumu se nachází na obrázku 3.18. Abychom předešli pohybu blesku

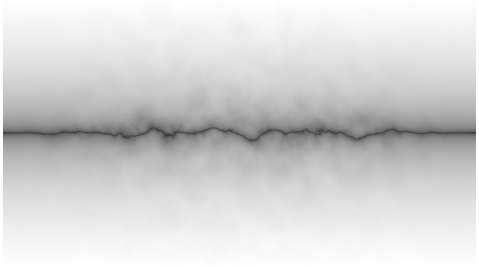


Obrázek 3.18: Kombinace simplexních šumů.

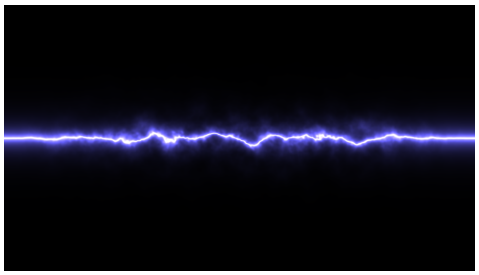


Obrázek 3.19: Aplikování šumu na y-ovou osu normalizovaných UV koordinátů v absolutní hodnotě

po celé šířce texury, kvůli návaznosti přes segementy, je tento pohyb omezen kvadratickou funkcí $-x^2 \cdot 0.15 + 0.15$. Výstup šumu je poté omezen pronásobením touto funkcí, aby nejvíce ovlivnil středovou část vygenerovaným šumem. Poté je k této hodně přičtena hodnota y-ové osy z normalizovaných UV koordinátů, abychom dostali gradientní přechod ze středu textury z černé do bíle. Tento přechod je vložen do absolutní hodnoty, aby vytvořila souměrný obrazec na středu osy y, jenž můžete vidět na obrázku 3.19. Pro získání jasnějšího obrazce na středu je potřeba hodnotu umocnit, nově vniklý obrazec naleznete na obrázku 3.20. Abychom obarvili daný obrazec, je potřeba hodnotou barvy pronásobit negativem obrazce a přičíst hodnotu barvy, v tomto případě musí být barva v rosahu $\langle 1, 2\rangle$. Výslednou barvu umocníme pro větší sytost barvy. Výslednou texturu naleznete na obrázku 3.21.



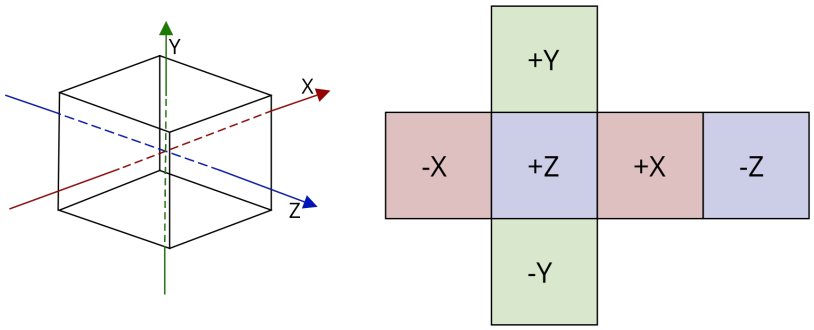
Obrázek 3.20: Výsledný negativ textury získaný umocněním hodnot z textury na obrázku 3.19



Obrázek 3.21: Výsledné obarvení.

3.7.7 Skybox

Pro vytvoření skyboxu, popsaný v kapitole 2.5, je potřeba nejprve provést namapování textur na krychli. Tato technika se nazývá cube mapping. Namapování jednotlivých textur bývá nejčastěji součástí grafického API, protože většina grafických karet dokáže hardwarově namapovat textury tak, aby s nimi dokázali pracovat jako s cube mapou. Při mapování jednotlivých textur se specifikuje strana krychle pomocí směru os. Obrázek 3.22 zobrazuje sestavení cube mapy podle os v OpenGL.



Obrázek 3.22: OpenGL Cubemapping.

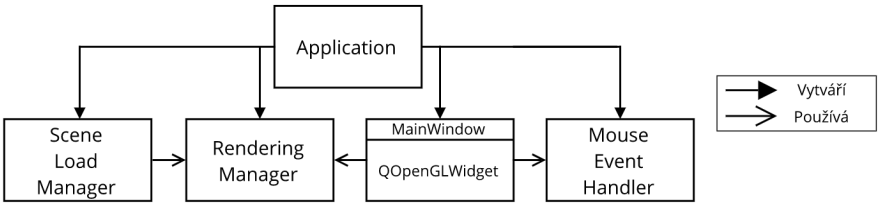
Hlavním rozdílem mezi vzorkováním textury a cube mapy je v adresování jednotlivých pixelů textury. U klasické textury se využívají 2D texturovací koordináty. Avšak u adresování cube mapy se používá normalizovaný směrový vektor. Pro názornost si lze představit paprsek, který je vystřilen ze středu krychle podle směrového vektoru, bod kde dojde k průniku paprsku s krychlí je navzorkován.

Většina moderní API dokáže vyřešit problém, kdy je vzorkován prostor mezi dvěma texturami. Ten nastává při sklonu směrového vektorů pod úhlem 45°, kdy se paprsek ocitne mezi dvěmi texturami. Aby došlo ke správnému navzorkování, tak se provede bilinearní interpolace mezi patřičnými texturami.

Kapitola 4

Implementace

Výsledná demonstrační aplikace je napsána v jazyce C++, společně s GPUEnginem a Qt5.3 pro vytvoření grafického rozhraní a zachytávání vstupů od uživatele. Skládá ze 4 hlavních částí, které zastřešuje statická třída `Application`. Hlavní třída pro grafické rozhraní je `MainWindow`. Další třídou je `RenderingManager`, který se stará o věci spojené s vykreslováním. S ním je spjatý `SceneLoadManager`, pomocí kterého se načítají modely s texturami. Poslední částí je `MouseEventHandler`, který se stará o zachytávání a zpracování událostí od uživatele.



Obrázek 4.1: Implementační diagram tříd.

4.1 Nahrávání scény

Hlavní třídou pro nahrávání scény je `SceneLoadManager`, který na svém vstupu přijímá třídu `RenderingManager`. Scéna je nejdříve nahrána pomocí addonu z GPUEnginu `AssimpModelLoader`. Poté jsou zkontrolované jednotlivé materiály. Pokud se jedná o materiál typu image, tak se jej pokusí addon `QtImageLoader` nahrát. V případě, že se obraz nepovede nahrát, je nahrazen prázdnou texturou. Nahraná scéna je dále předána `RenderingManageru` pro další zpracování.

4.2 Grafické rozhraní

Ústřední třídou pro grafické rozhraní je `MainWindow`, která pomocí Qt knihovny vytvoří grafické okno spolu s třídou `ApplicationOpenGLWidget`. Třída `ApplicationOpenGLWidget` je zděděná z třídy `QOpenGLWidget`, která vytváří základní vykreslovací smyčku v Qt. Zde probíhá inicializace GPUEnginu a vytvoření OpenGL kontextu. Na tento widget je v rámci třídy `Application` navázán filtr pro zpracování událostí vyvolaných myší, ve kterém se tato akce zpracuje.

4.3 Zpracování událostí

Na zpracování událostí byla využita možnost implementace filtru, kterou nabízí Qt. V aplikaci nebylo nutné zpracovávat vstupy z klávesnice, takže třída implementující zpracování událostí se nazývá `MouseEventHandler`. Pokud je třída implementována jako filtr událostí, tak musí naimplementovat metodu `eventFilter`, jenž vrací boolovskou hodnotu. Pokud je tato návratová hodnota true, tak se událost nepřeposílá na další objekty, ale je považována za zpracovanou. Jestliže se událost nevyfiltruje, tak je následovně předána rodičovské třídě.

Pro potřeby této třídy jsou nadefinovány 2 datové struktury. Jedna pro pohyb myší a druhá pro aktuálně aktivní tlačítka myši. Rotace kamery je řešena pomocí levého tlačítka myši, kdy je při stisknutí zapamatována x, y pozice v rámci `OpenGLWidgetu`. Následně na to každý pohyb myší vyvolá událost o pohybu, při kterém je odečtena aktuální pozice od staré pozice v poměru k šířce nebo výšce okna. Výsledek je poté přičten k úhlu kamery, který se projeví v pohledové matici. Posouvání kamery ve scéně je řešeno stejným stylem jako rotace, pouze musí být aktivní pravé tlačítko myši. Oddalování se počítá z pohybu kolečka myši a následně je přičteno k aktuální vzdálenosti kamery.

4.4 Vykreslovací smyčka

Hlavní vykreslovací smyčka je v `ApplicationOpenGLWidget`. V této smyčce jsou volány metody z `RenderingManageru`. Ten obsahuje vše spojené s výkreslováním do OpenGL kontextu, udržuje si kamery, čas v rámci scény a všechny vizualizační techniky spolu s manažery objektů. Z návrhového hlediska implementuje vykreslovací část aplikace. Fáze ve které dochází k přípravě scény byla rozdělana do dvou objektů: `SceneLoadManager` a `RenderingManager`.

`SceneLoadManager` se stará o nahrání objektů do datové struktury `Scene`. `RenderingManager` vytváří kameru společně s manažery objektů a vizualizačními technikami. Nastaví animace pro kameru a modely lodí, které jsou popsány v kapitole 3.4. Manažerům objektů zařídí spojení s animačním manažerem a předá jim potřebné informace, jako je čas nebo kamera. Vizualizačním technikám zkompiluje a slinkuje jednotlivé GLSL programy, které jim následně předá společně s OpenGL kontextem pro vykreslování.

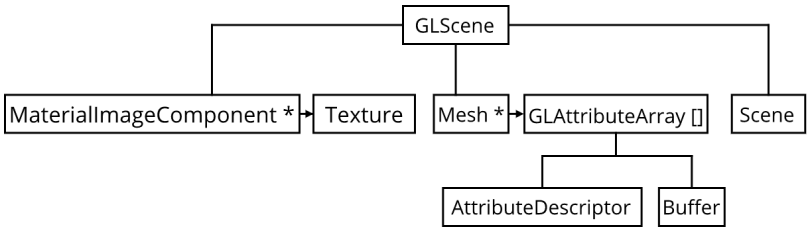
Čas se v rámci scény měří pomocí `high_resolution_clock` ze standartní knihovny C++. V inicializační fázi se zapamatuje čas startu aplikace. Pro získání aktuálního času v aplikaci pak stačí pouze zjistit aktuální hodnotu času a odečíst ji od času startu aplikace. Převod na vteřiny probíhá pomocí templátované funkce `duration_cast`.

Ve své vykreslovací smyčce se zkontroluje, zda je potřeba zpracovat nahranou scénu a aktualizuje čas scény. Ten se předá animačnímu správci pro spočítaní jednotlivých interpolací pro animace. Manažeři zkontrolují, zda je potřeba renderovat všechny instance objektů, které mají uloženy. Proběhne kontrola, zda nedošlo k průniku střel se štítem v průběhu animace. Nakonec se provede aktualizace uniformních proměnných, jako jsou projekční a pohledová matice a čas.

4.4.1 Zpracování načtených modelů

Modely načtené v datovém kontejneru `Scene` je potřeba převést na datové struktury spojené s OpenGL. Tuto funkcionalitu provádí třída `GLSceneProcesor`, která byla převzata z příkladů v GPUEnginu spolu s třídou `GLScene` a `TextureFactory`. Třída `GLScene` je datová struktura sloužící pro namapování meshe a obrázku na datové struktury spojené s OpenGL.

`GLSceneProcesor` provádí převod nad těmito datovými strukturami ve 2 fázích. Nejdříve jsou namapovány všechny meshe objektu na `GLAttributeArray`, které se skládají z popisu dat a OpenGL bufferu. Protože všechny vlastnosti jednotlivých bodů jsou poskládány v paměti lineárně za sebou, je potřeba uchovávat informace o tom, jak jsou jednotlivé vlastnosti uloženy v této paměti. Tyto informace jsou nadále použity při nastavování Vertex Pulleru pro Vertex Array Object. Dalším krokem je namapování materiálových komponent, které obsahují obrázky pro textury. Převod obrázku na textury probíha pomocí třídy `TextureFactory`, která převezme `MaterialImageComponent` s potřebnými informacemi pro vytvoření textury.



Obrázek 4.2: Reprezentace scény pro OpenGL.

Vizualiační technika

Vizualizační technika 3.2, pro klasické meshe, provádí poslední krok převodu do openGL. Pro každý mesh vytvoří Vertex Array Object, do kterého se postupně předávají vytvořené OpenGL buffery spolu s informacemi poskytovanými attribute descriptory. Každý VAO (Vertex Array Object) je poté namapován na mesh, kterému přísluší. Dále jsou zpracovány materiály. Jako první se zpracuje difúzní barva, následně jednotlivé textury, které jsou namapovány na mesh.

Při vykreslování se prochází jednotlivé meshe, pro které se nastaví difuzní barva do uniformní proměnné. Pomocí namapovaných meshu se sváže textura a VAO s grafickou kartou. Shadery pro vizualizaci modelu jsou převzaté z příkladů v GPUEnginu. Vertex shader provádí přesun modelu na správné místo pomocí transformačních matic a Fragment shader provádí nahrání textury na objekt či obarvení pomocí difúzní barvy. Výsledek vykreslení modelu je na obrázku 4.3.

4.4.2 Manažeři

Pro jednotlivé efekty jsou navrhnuté pouze základní třídy pro uchovávání dat a vizualizační techniky. Proto se zavádějí manažeři, kteří zastřešují práci nad daným efektem. Udržují si všechny instance obecných popisů objektu, provádějí operace jako přidání či odstranění prvku. Někteří manažeři si udržují informace o animacích, které jsou uložené v animačním správci. Při mázaní prvku je potřeba smazat i s ním spjaté animace, aby nedošlo k interpolaci nad zaniklým objektem. Manažeři také předávají všechny uložené instance vizualizační technice, která následně převede objekty do OpenGL reprezentace vhodné pro vykreslení.

4.4.3 Lasery

Laser je reprezentován pomocí 2 bodů v prostoru, barvy a modelovou maticí. Lasery se sdružují v `LaserManageru`, který nad nimi udržuje kontrolu a implementuje animace pro

Obrázek 4.3: Vykreslený model.

trajektorii letu a nárazu střely, popsaného v kapitole 3.6. Pro animaci trajektorie letu je vytvořen animační kanál nad modelovou maticí. Tento kanál se skládá ze dvou klíčových snímků, přičemž první je definován v čase nula a druhý o deset vteřin později. V prvním klíčovém snímku je nastavena aktuální pozice laseru a ve druhém se k aktuální pozici přičte násobek rychlosti a normalizovaného směrového vektoru. Při nárazu laseru do štítu se vytvoří animace nad počátečním bodem laseru, který během půl vteřiny nabude hodnoty koncového bodu a zkrátí tak svou délku na nula. Při přidání laseru do manažeru se vytvoří animace letu, která je namapována na laser z důvodu odstranění laseru před ukončením animace. Hned poté je animace předána animačnímu správci s aktuálním časem pro spuštění. V aktualizační smyčce probíhá kontrola, zda je potřeba laser nadále uchovávat v paměti. Důvody k odstranění mohou být následující: Laser již dokončil svou animaci nárazu anebo je laser velice vzdálen od kamery.

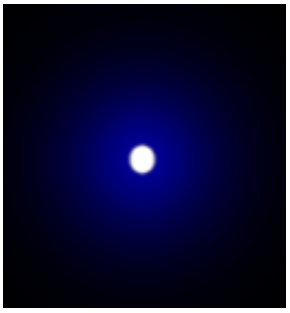
Vizualiační technika

Vizualizační technika, pro lasery, si nejdříve ve své aktualizační části aktualizuje pozici kamery, která hraje klíčový prvek při vytváření laseru. Poté jsou vymazány všechny body uložené z předchozího vykreslení, z důvodu pohybu kamery nebo pohybu samotných laserů ve scéně, aby bylo dosaženo správného natočení billboardu 2.4. Následně se začnou počítat koordináty billboardů pro každý laser, vysvětlené v sekci 3.7.3. Pro všechny lasery je následně vytvořen VAO, kde se nahrají všechny body zpracovaných laserů. Ve vykreslovací části je potřeba zapnout směšování a vybrat pro něj funkci pro dosažení transparentosti billboardu v části, kde se nenachází laser. Vytvořený VAO spolu s texturou se sváže s grafickou kartou. Vybere se GLSL program 3.5 a lasery se vykreslí. Při vykreslování je ve Fragment Shaderu provedena projekce texturovacích koordinátů na texturu z důvodů popsaných v kapitole 3.7.3 a následné obarvení na zadanou barvu. Výstup této vizualizační techinky lze naleznout na obrázku 4.4.

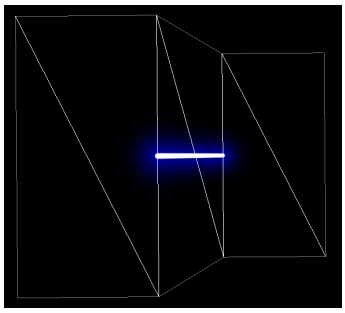
4.4.4 Energetické střely

Energetické střely se od své třídy pro geometrickou reprezentaci liší pouze přidáním modelové matice a směrového vektoru, který určuje trajektorii letu střely. Také implementuje

(a) Boční pohled.



(b) Čelní pohled



(c) Boční pohled s geomterií.

Obrázek 4.4: Vizualizace laseru.

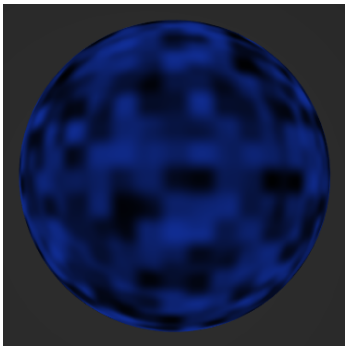
metody pro zjištění střetu se štítem, popsané v kapitole 3.6. Vytvořená energetická střela musí být předána `EnergyBallManageru`, který všechny uchovávané instance předává vizualizační technice. Stejně jako u laseru, je při přidání energetické střely do manažeru, vytvořená animace společně s namapováním na danou střelu. Trajektorie střely se vypočítá pomocí počátečního bodu a směrového vektoru. Při nárazu do štítu se spouští animace postupného zmenšování koule až do nulové velikosti. Manažer pak ve své aktualizační části zkontoroluje, zda nedošlo k dokončení animace střetu a nebo zda je v dostatečné vzálenosti od kamery, z důvodu odstranění.

Vizualizační technika

Vyzualizační technika, pro energetické střely, si při své inicializaci vytvoří jednu kouli pomocí třídy `SphereFactory`, která implementuje vytvoření geometrie koule, popsané v podkapitole 3.7.1. Pro tuto kouli je rovnou vytvořen VAO, který se používá ve vykreslovací části. Při vykreslení se vytvoří pole modelových matic pro jednotlivé střely. Ty jsou následně svázany s grafickou kartou jako SSBO (Shader Storage Buffer Object). Vybere se program pro vykreslování energetických střel a ty jsou následovně vykresleny pomocí příkazu `glDrawArraysInstanced`. Díky tomuto příkazu se dostává do Shaderu proměnná `gl_InstanceID`, pomocí které je vybrána správná transformační matice pro každou instanci. Ve Fragment shaderu je poté aplikován 3D Perlinův šum 2.2, který je obarven na modro-černou barvu a použit jako textura pro střelu. Výsledek vizualizační techniky můžete vidět na obrázku 4.5.

4.4.5 Štíty

Štíty si udržují svoji geometrii a modelovou matici společně s maticí objektu, na který jsou navázány. Také si uchovávají informace o střetu střel se štítem pro efekt propalující se střely a pulzní vlny, které se aktuálně nachází na štítu. Všechny vytvořené štíty se ukládají do svého manažeru, který umí vytvářet animace nad parametry t obsažené jak v pulzní vlně,



Obrázek 4.5: Vizualizace energetické střely.

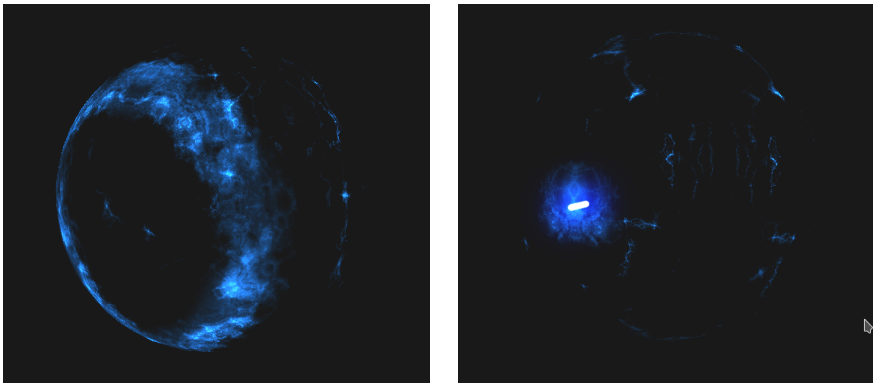
tak při efektu propalující se střely štítem. Při přidávání štítu do manažeru je vytvořena koule společně s modelovou maticí, která obsahuje zvětšení podle zadaného poloměru a pozici ve scéně.

Ke třídě `ShieldManager` se navíc váže třída `ShieldIntersector`, která v aktualizační části zjistí, zda došlo od minulého snímku k novému nárazu do štítu. Pokud dojde k průniku střely se štítem, tak zastaví animaci letu střely a spustí animaci zániku střely, které jsou již popsány výše u jednotlivých střel. Následně na to spočítá bod střetu, který předá `ShieldManageru`, jenž vytvoří a následně spustí potřebné animace pro jednotlivé efekty a sváže je s příslušnou instací štítu.

Vizualizační technika

Vizualizační technika, pro štíty si stejně jako energetické střely, vytvoří při své inicializaci jednu jednotkovou kouli, pro kterou vytvoří VAO. Pro vytvoření průhledných štítů je potřeba zapnout směšování barev spolu se směšovací funkcí pro transparentnost. Abychom mohli vykreslit všechny štíty jedním zavoláním vykreslovací funkce, je potřeba vytvořit pět vektorů pro: matice, průniky, pulzní vlny a indexy pro průniky a pulzní vlny. Poté se postupně prochází jednotlivé štíty a ukladájí se počty průniků a pulzních vln pro indexy, matice a informace z průniků a pulzních vln. Nad uloženými počty průniků a pulzních vln se provede algoritmus parciálních sum, který pro N-té místo v poli uloží hodnotu součtu všech předchozích členů včetně aktuálního člena. Po vykonání tohoto algoritmu se nám vytvoří indexy, které odpovídají uloženým informacím pro danou kouli. Aby bylo možné provádět dopřednou indexaci, je na první místa vložena nula. To zajistí, že informace pro specifickou kouli N, nalezneme podle indexu uložených v indexovacích polích, informace se potom prochází z intervalu $\langle I_P[N], I_P[N+1])$, kde $I_P[N]$ je číslo v indexovacím poli, příslušící danému efektu, na místě N. Po uložení veškerých informací se nastaví program pro grafickou kartu společně s VAO, který má v sobě uložené body jednotkové koule. Pro všechna vytvořená pole jsou vytvořeny SSBO, které se sváží s grafickou kartou. V tento moment jsou připravené všechny informace pro vykreslení. Vykreslení je spuštěno příkazem `glDrawArraysInstanced`, aby bylo možné vyhledávat v polích pomocí proměnné `gl_InstanceID`.

Ve Vertex shaderu jsou štíty umístěny na správné koordináty pomocí transformačních matic. Fragment shader vytvoří pro štíty fraktálovitou texturu popsanou v kapitole 3.7.5. Tento fraktál je ztmaven. Pomocí masek vytvořených pro pulzní vlnu nebo efekt propalující se střely, popsaných v kapitolách 3.7.5 a 3.7.5, je zesvětlován. Vypočítaný fragment je nakonec obarven do modré barvy. Výsledný štít lze nalézt na obrázcích 4.6.



(a) Pulzní vlna (b) Efekt propalování střely.

Obrázek 4.6: Vizualizace štítu.

4.4.6 Elektrický oblouk

Jednotlivé části elektrického oblouku se skladají ze struktury `LightningSegment`, která si ukládá počátek a konec segmentu společně s jeho hloubkou z důvodu větvení. Samostatný oblouk je definován svým počátečním bodem a délkou. Pro vygenerování oblouku slouží statická třída `LightningFactory`, která je schopna vygenerovat geometrii popsanou v kapitole 3.7.6. Pro pseudonáhodná čísla využívá generátor mt19937, který je jedním z Mersenne Twister generátorů popsaných v kapitole 2.1.2. Následně se vygenerované úsečky převedou na line strip a oblouk je pro vizualizační techniku označen jako nezpracovaný.

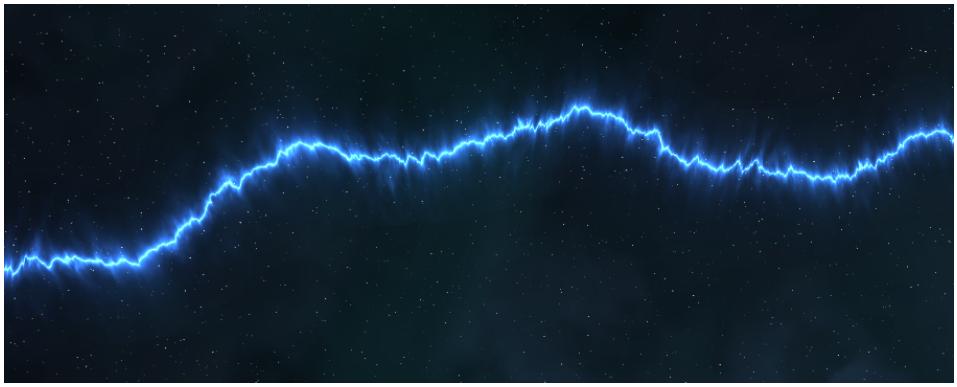
Vizualizační technika

Ve vizualizační technice se v akutalizační části zkontroluje, zda byl oblouk již zpracován na primitivum `GL_QUAD`. Pokud ne, tak se pro první a poslední body spočítají body billboardu společne s uv koordináty na CPU. Zbytek dat je odeslán na GPU pomocí SSBO, kde je v Compute shaderu spočítaná návaznost jednotlivých billboardů, popsaná v kapitole 3.7.6. Po přepočtu je pro oblouk vytvořený VAO, do kterého se uloží vypočítané hodnoty a ten je pak namapován na instanci oblouku pomocí `unordered_map`. Toho je využito při vykreslování, kdy je iterováno přes tohle namapování a postupně se nastavují modelové matice společně s namapovaným VAO pro vykreslení a ve smyčce se po jednom vykreslují. Ve Vertex shaderu je potom využita pohledová matice, pomocí které je vytvořen cylindrický billboarding. To znamená, že se oblouk otáčí pouze okolo osy Y. Ve Fragment shaderu se poté vytváří textura popsaná v kapitole 3.7.6. Velká část shaderu byla převzata z webu[1]. K převzanému shaderu byla přidána animace textury pomocí Perlinova šumu, který se přičítá při omezení textury na UV y-ové ose. Výstup vizualizační techniky můžete vidět na obrázku 4.7.

4.4.7 Skybox

Pro vytvoření skyboxu, popsaného v kapitole 3.7.7, je nutné vytvořit CubeMap texturu. Tuto funkcionalitu obstarává třída `CubeMapTexture`, ve které se načtou jednotlivé tex-

[1] https://www.shadertoy.com/view/4scGWj



Obrázek 4.7: Vizualizace elektrického oblouku.

tury. Následně v inicializační funkci třídy `CubeMapTexture` je vytvořena CubeMapa pomocí OpenGL funkce `glTexutureSubImage2DEXT`.

Ve vizualizační technice je pomocí dekompozice pohledové matice zjištěna orientace, která je následovně pronásobená inverzní perspektivní maticí a poslána na grafickou kartu společně s velikostí OpenGLWidgetu. Vertex shader je v této technice prázdný, následný Geometry shader vytvoří jednotkovou krychli. Ve Fragment shaderu je následově spočítána pozice fragmentu v okenních koordinátech. Těmito koordináty se pronásobí inverzní matice, která byla poslána na grafickou kartu jako uniformní proměnná, čímž získáme aktuální vektor pohledu kamery. Pro jednotlivé fragmenty je pak navzorkována cube mapa pomocí aktuálního pohledu. Výsledek použitého skyboxu v demonstrační aplikaci lze vidět na obrázku 4.8.

Obrázek 4.8: Použitý skybox.

Kapitola 5

Závěr

Cílem této práce bylo vytvoření knihovnu pro procedurálně generované efekty. Při vytváření knihovny bylo zapotřebí se seznámit s knihovnou OpenGL, se kterou jsem neměl skoro žádné předešlé zkušenosti a nastudovat základní techniky pro procedurální generování grafického obsahu. V demonstrační aplikaci jsou využity veškeré efekty, které knihovna nabízí: elektrický výboj, štíty, lasery a energetické střely.

Lasery jsou tvořeny třemi billboardy, které se smršťují nebo rozšiřují podle úhlu svíraného mezi kamerou a laserem. Následně je na ně pomocí projektivních koordinátů navzorkována textura. Štíty jsou tvořeny primárně pomocí textury fraktálovitého charakteru, přičemž se k němu váží ještě dva efekty. První je efekt propalující se střely skrz štíty. Toho je docíleno gradientní maskou v místě nárazu, která je přičtena k intezitě viditelnosti štítu. Druhým je pulzní vlna, která se objeví při nárazu a postupně projede celým štítem. Tento efekt je tvořen maskou prstencovitého tvaru, která postupně přechází přes štít. Energetická střela, jejíž geometrie je vytvořena pomocí goniometrických funkcí a textura Perlinovým šumem. Elektrický výboj se skládá z několika úseček, jenž jsou pomocí Compute shaderu převedeny na řadu navazujících billboardů, na které se pomocí kombinace Perlinova a Simplexního šumu vytváří textura.

Jako další rozšíření se nabízí implementace volumetrických efektů nebo využití částicových systému pro nový druh efektů. Pro vylepšení stávajících efektu by bylo vhodné přenést výpočet billboardů laseru do Geometry shaderu nebo vytvoření efektu blesku, pro který je již napsaný algoritmus generování geometrie. V případě elektrického výboje by mohlo být předěláno cylindrické billboardování na rotaci billboardu podle své vlastní osy.

Literatura

[1] Boiangiu, C.-A.; Gabriel Morosan, A.; Stan, M.: Fractal Objects in Computer Graphics. 06 2015.
URL https://www.researchgate.net/publication/287218131_Fractal_Objects_in_Computer_Graphics

[2] Ebert, D. S.; Musgrave, F. K.; Peachey, D.; aj.: *Texturing and Modeling: A Procedural Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., třetí vydání, 2002, ISBN 1558608486.

[3] Gustavson, S.: Simplex noise demystified. *Linköping University, Linköping, Sweden, Research Report*, 2005.

[4] Matsumoto, M.; Nishimura, T.: Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.*, ročník 8, č. 1, Leden 1998: s. 3–30, ISSN 1049-3301, doi:10.1145/272991.272995.
URL http://doi.acm.org/10.1145/272991.272995

[5] O'Neill, M. E.: PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. September 2014, [Online].
URL http://www.pcg-random.org/pdf/hmc-cs-2014-0905.pdf

[6] Perlin, K.: An Image Synthesizer. *SIGGRAPH Comput. Graph.*, ročník 19, č. 3, Červenec 1985: s. 287–296, ISSN 0097-8930, doi:10.1145/325165.325247.
URL http://doi.acm.org.ezproxy.lib.vutbr.cz/10.1145/325165.325247

[7] Perlin, K.: Noise hardware. *Real-Time Shading SIGGRAPH Course Notes*, 2001.

[8] Perlin, K.: Improving Noise. *ACM Trans. Graph.*, ročník 21, č. 3, 7 2002: s. 681–682, ISSN 0730-0301, doi:10.1145/566654.566636.
URL http://doi.acm.org/10.1145/566654.566636